Nuts and Bolts of Deploying Real Cloudproxy Applications

John Manferdelli[1], Sidarth Telang

Overview

Cloudproxy is a software system that provides *authenticated* isolation, confidentiality and integrity of program code and data for programs even when these programs run on remote computers operated by powerful, and potentially malicious system administrators. Cloudproxy defends against observation or modification of program keys, program code and program data by persons (including system administrators), other programs or networking infrastructure. In the case of the cloud computing model, we would describe this as protection from co-tenants and data center insiders. To achieve this, Cloudproxy uses two components: a “Host System” (raw hardware, Virtual Machine Manager, Operating System) which provides capabilities described below to the protected program or “Hosted System” (VM, Application, Container).

This document focuses on installation, security and deployment tradeoffs for Cloudproxy applications. In addition, we explain at length several options for providing key management and program operation continuity as programs are upgraded or new programs are introduced. Programmers new to Cloudproxy often worry that program upgrade is impossible or difficult but it is not. We also describe successful key management strategies which support frequent key rotation which is strongly recommended to ensure ongoing security as standard practice as well as providing resilience in the face of errors or omissions.

We hope these instructions allow safe and rapid installation and configuration of Cloudproxy nodes without extensive training or preparation. We assume readers are familiar with [4] which explains Cloudproxy basic concept like measurement and principal names and explains how to build Cloudproxy applications.

Readers can consult [1] for a fuller description. Source code for Cloudproxy as well as all the samples and documentation referenced here is in [2].

Downloading and compiling Cloudproxy

First, you should download the Cloudproxy repository from [2]. To do this, assuming you have git repository support, type

    git clone https://github.com/jlmucb/cloudproxy,

or,

    go get https://github.com/jlmucb/cloudproxy.

[1] John isat manferdelli@google.com or jlmucbmath@gmail.com; Sid is at sidtelang@google.com. Cloudproxy is based on work with Tom Roeder (tmroeder@google.com), Fred Schneider (Cornell) and Kevin Walsh (kevin.walsh@holycross.edu).

This latter command will also install the needed go libraries. You can also download a zipped repository from github. You should probably install this in ~/src/github.com/jlmucb (which we refer to as `$CLOUDPROXYDIR`) to save go compilation problems later. It's a good idea to put go binaries in `~/bin` as is common in Go. Follow the installation instructions in `$CLOUDPROXYDIR/Doc`; that directory also contains [1] and an up to date version of [4].

You must also install the Go development tools (and C++ development tools if you use the C++ version) as well as protobuf, gtest and gflags as described in the Go documentation.

We will continue to use the simpleexample application described in [4].

Security Model

Any deployment of a security system must consider the security or threat model that a system seeks to provide. For the purpose of this Deployment Guide, we consider you are running software for your clients in a cloud data center (or other shared facility) and storing your data there.

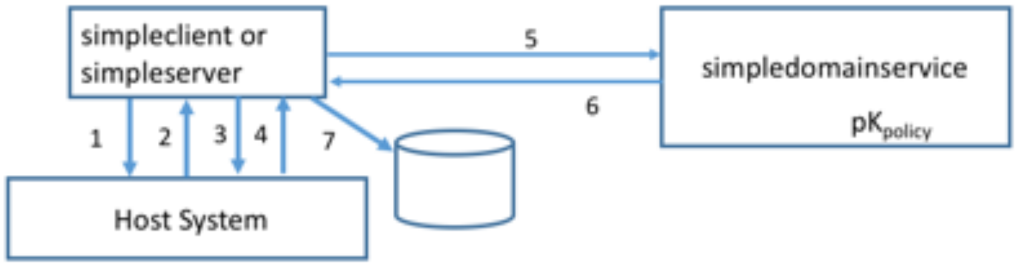
Your goal is to provide the principled, verifiable *confidentiality and integrity* of the execution of your programs and the data they read and store from unrestricted attacks from outsiders (e.g. – other clients of the cloud provider) as well as attacks from individual insiders in the cloud data center.

We do assume that data center insiders do not have physical access to computers running your software while that software is running. If they did, insiders could mount physical attacks on running hardware, for example, attaching memory probes to buses. This level of protection can be ensured by, say, enclosing the computers running client software in locked cages that prevent physical access to designated computers while the computers are running Cloudproxy, and for a few minutes afterwards (to protect against insiders harvesting memory remnants of running programs). An extra measure to ensure compliance might be cameras monitoring the caged systems. We *do not* require all computers or networking equipment to be in such cages. In fact, storage systems and networking equipment could remain unprotected without violating the foregoing security guarantee.

We also require that the software you provide (or use) not have remotely exploitable vulnerabilities but we *do not* limit what software you can run or significantly change the common programming model.

Overview of installation on Linux

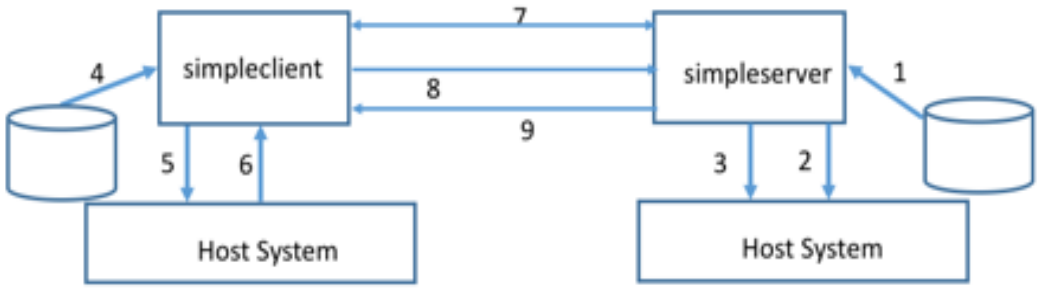
The figure below should be familiar from [4]. We will refer to this throughout the document.



Initialization

1. simpleclient (or simpleserver) generates public/private key pair $PK_{simpleclient}$, $pK_{simpleclient}$. simpleclient requests Host System attest $PK_{simpleclient}$.
2. Host System retuns attestation
3. simpleclient generates additional symmetric keys and request Host System seal symmetric keys and $pK_{simpleclient}$.
4. Host System returns sealed blobs.
5. simpleclient connects to simpledomainservice and transmits attestation.
6. simpledomainservice returns signed Program Certificate.
7. simpleclient stores sealed blobs and Program Certificate for later activations.

Initialization



Operation

1. Simpleserver reads previous sealed blobs and Program Certificate.
2. Simpleserver requests Host System unseal blobs yielding symmetric keys and private program key.
3. Host system returns unsealed blobs.

4. Simpleclient reads previous sealed blobs and Program Certificate.
5. Simpleclient requests Host System unseal blobs yielding symmetric keys and private program key.
6. Host system returns unsealed blobs.

7. Simpleclient and simpleserver open encrypted, integrity protected channel using their program keys and certificates.
8. Simpleclient transmits a request to retrieve secret.
9. Simpleserver retruns secret.

Operation

All the hardware enforced mechanisms described in this document are based on the primary system software being Linux based. This includes a native Linux booted on an Intel SMX/TPM platform as well as the KVM hypervisor. Two configurations we discuss (a hosted Linux stacked on KVM and a Docker based Tao stacked on a Linux Tao) do not run directly on hardware but are hosted by a lower level Linux based Tao. We refer to a Cloudproxy Host booted on hardware as a Base Cloudproxy Host. The comments in this section thus refer to a KVM base system or a native Linux Base Cloudproxy system. With fairly minor changes, one could also make an BSD based platform a Base Cloudproxy Host.

The need to perform a measured requires a little more software support. Linux Base Cloudproxy Host use a multi-boot sequence to load the Linux kernel mediated by Grub. The stack of booting software consists of:

1. Grub which loads the other booting software into memory.
2. Tboot which mediates the authenticated boot using SMX and the TPM.
3. The Linux kernel with TPM support.
4. A custom initramfs which is measured as part of the boot and contains the compiled Cloudproxy components and critical libraries. Unlike the normal boot sequence, the initram filesystem remains the root file system throughout the base system's activation.

All other file systems are mounted under it. Note that all such mounted file systems (as well and any network connected storage) are treated as untrusted by Cloudproxy programs, and files are encrypted and integrity protected for reading and writing.

5. Dmcrypt which is used to encrypt the page-file and swap space[2] (so no unencrypted data is stored on the paging disk).

We describe the role and general installation procedure for each of these below. For each, there is also an appendix with detailed installation instructions. In addition, we describe preparing, measuring and running Linux based VM's with Cloudproxy support which can be hosted by KVM.

The overall procedure for preparing and installing Cloudproxy root system is:

1. Install Linux. TPM2.0 requires a Version 4 kernel or later. The kernel should contain the appropriate TPM driver and *dmcrypt*. We remark that, for security reasons, the TCB of the root Cloudproxy system should be as small as possible. A carefully constrained kernel is a critical element of security. For example, our Linux root Cloudproxy systems either disallow loading dynamic modules or subject proposed loads to a small whitelist of signed modules. The base Cloudproxy should include *Dmcrypt*.
2. Install the Linux Kernel Development environment, on development machines.
3. Enable SMX, IOMMU and VT extensions in BIOS. Activate the TPM using the BIOS in the case of TPM 2.0 or using trousers in the case of TPM 1.2.
4. Take ownership of the TPM and generate the appropriate keys.
5. Build TBOOT. Retrieve the required Authenticated Code Module (ACM) required by Tboot and place it and the tboot image in /boot as described below.
6. Build the Cloudproxy binaries. Then build a custom initramfs with these binaries as described below. Put the newly created initramfs in /boot/
7. Configure Grub. You will boot a grub configuration which names Tboot, the ACM, your Linux kernel and your custom Initramfs.
8. Boot!

The "measurement" of a TPM mediated boot will be in PCR 17 and 18 of your TPM. You will need this measurement for your domain service and the easiest way to get it is to have Cloudproxy report it after it boots on your test hardware. Cloudproxy does the rest! After you have successfully carried out these steps, you can build applications as described in [4].

Next, we describe each of these steps in more detail. After that, we discuss secure, scalable deployment of Cloudproxy applications including key management techniques followed by a brief description of Security Considerations which should be thought through for your particular deployment needs.

---

[2] A subtle point: Dmcrypt at the time of this writing, encrypts but does not provide integrity protection. As a result, the "paging disk" (which might be flash or spinning media) should be in the "physical protection barrier" described in the Security Considerations section of this document.

Hardware root of trust

Because of a lack of physical control over the hardware (and even with physical control in the face of errors), maintaining our security model requires a hardware root of trust that is used to measure the base system software that boots and ensure that only software and only software trusted by you can access some basic system secrets like signing keys. Without this, insiders could boot any privileged software and violate the security model you seek to enforce. Cloudproxy, as mentioned above can employ TPM1.2 or TPM 2.0 as “off the shelf” mechanisms. So first you must enable and provision one of these.

BIOS

In Bios, you must ensure that VT and IOMMU support is enabled. The Tpm must also be enabled (usually it is enabled by default in the case of TPM 2.0 but is disabled by default with TPM 1.2).

Trusted Platform Module (TPM)

The TPM is a dedicated microprocessor designed to offer facilities for generation of cryptographic keys, random number generation, remote attestation and data sealing. In this document, we use the TPM as the hardware root of trust of our Cloudproxy system.

To use the TPM we need to enable it, and take ownership of it. Taking ownership of a TPM is the process of setting up an authorization value (e.g. – a password), which is subsequently required for certain kinds of TPM operations.

Taking ownership of a TPM is the process of setting up authorization values (eg. passwords) for the *ownerAuth*, *endorsementAuth*, and *lockoutAuth* types of authorization. Different TPM commands require different types of authorization. A cleared TPM (using TPM2_Clear) has these values set to *emptyAuth.*

Here are instructions for TPM 1.2 and TPM 2.0.

TPM 1.2

There are several keys in a TPM 1.2, which include:

1. TPM Endorsement Key (EK): This key is created by the manufacturer and cannot be removed. Sometimes it can be changed by the owner of the computer.

2. TPM Storage Root Key (SRK): Is the 2048 bit RSA key created when configuring the ownership. This key is stored inside the chip and can be removed. This key is also used to seal/unseal data.
3. TPM Attestation Identity Key (AIK): This key is used by the TPM to sign attestations (to the state of the PCRs). It is encrypted and integrity protected by the SRK.

Cloudproxy uses the SRK to seal/unseal data and the AIK to sign attestations. However, all we need to get Cloudproxy running is an AIK blob. Following are instructions on how to obtain it, and to provision the TPM for use by Cloudproxy.

1. First, we install tools needed to talk to the TPM device.
   ```
   go get -u -v github.com/google/go-tpm/…
   ```

2. Reset your TPM device as follows.

   - Disable the TPM device on your BIOS screen.
   - Reboot the system **by powering off**, re-enable the TPM device in BIOS, and select the option to clear the device if you see one.

   TPM devices require an assertion of physical presence to be cleared, and for most machines powering off and on does the trick. But ultimately how presence is asserted depends on your machine, so consult your BIOS manual for detailed instructions.
   Also, note that the TPM device comes disabled by default.

3. Take ownership of the device, which may need root privileges. Note that tpm-takeownership takes the new owner and SRK password from the environment, and that we do not set them. The current Cloudproxy implementation assumes these authorizations are not set, when talking to the TPM.
   ```
   unset TPM_OWNER_AUTH
   unset TPM_SRK_AUTH
   sudo su --preserve-environment
   tpm-takeownership
   ```

4. Generate the aikblob and set permissions so that Cloudproxy may read it (note that this blob is encrypted and integrity protected by the TPM so it can be made public).
   ```
   genaik
   chmod 777 aikblob
   ```

Some troubleshooting tips can be found in the appendix.

TPM 2.0

Taking ownership of a TPM 2.0 is the process of setting up authorization values (eg. passwords) for the *ownerAuth*, *endorsementAuth*, and *lockoutAuth* types of authorization. Different TPM 2.0 commands require different types of authorization. A cleared TPM (using TPM2_Clear) has these values set to *emptyAuth.* To use Cloudproxy, we need to set each of the above authorization to *emptyAuth* (i.e. empty string as a password).

Enabling, taking ownership of and clearing a TPM 2.0 device is done entirely through BIOS. Consult your BIOS manual for detailed steps. Detailed instructions for the Intel NUC 5i5MYHE can be found in the appendix.

Tboot

Trusted Boot (tboot) is a pre-kernel/VMM module that uses Intel Trusted Execution Technology (Intel TXT) to perform a measured and verified launch of an OS kernel/VMM. It uses the TPM for this and when successfully run, it stores the kernel/VMM measurement in the TPM PCRs, which is then read by the root Cloudproxy Tao.

Installation overview: After making sure your TPM device is enabled (see previous sections), download the tboot source tarball from sourceforge.net, and build and install using `make` and `sudo make install`. This build a tboot.gz image and places it in /boot for your bootloader to use when booting up. You will also need a SINIT AC module for your type of processor that you must manually place in /boot. This module enables TXT to load tboot. Finally you need to update your bootloader configuration to load tboot. Detailed instructions are in the appendix.

Some miscellaneous but probably useful notes on tboot follow.

txt-stat: The txt-stat tool installed by tboot is a Linux application that reads some of the TXT registers and will display the tboot boot log if tboot was run with 'logging=memory'. It is useful to run after a boot-up to check if tboot was run successfully.

Launch control policy (LCP) and Verified Launch Policy (VLP): These are policies that govern which measured launch environments (MLEs, for eg. tboot) and kernels TXT and tboot are allowed to launch respectively. Tboot comes with a set of tools to install LCPs and VLPs, however they are not needed for Cloudproxy.

PCR values after trusted boot:

PCR 17 :

It will be extended with the following values (in this order):

- The values as documented in the MLE Developers Manual
- SHA-1 hash of: tboot policy control value (4 bytes) |
  SHA-1 hash of tboot policy (20 bytes)

  : where the hash of the tboot policy will be 0s if TB_POLCTL_EXTEND_PCR17 is clear

PCR 18 :

It will be extended with the following values (in this order):
- SHA-1 hash of tboot (as calculated by lcp_mlehash)
- SHA-1 hash of first module in grub.conf (e.g. Xen or Linux kernel)

Detailed copy and paste instructions are in the appendix.

Preparing the Linux or KVM image

Linux systems need to be properly configured to meet our security goals. First, on most images, there should be no way for someone to log on and become root (for example, via SSH). Becoming root in a Cloudproxy image means the person doing so can violate policy since they can run any program or inspect any memory. On most Linux systems there should be no interactive users at all and any administrative tasks should be carried out using constrained, authenticated interfaces. The kernel should be as small as possible and no one should be able to load kernel modules --- i.e. any device driver should be examined and compiled into the original kernel image. Linux images should be carefully configured, maintained, curated and inspected.

We also need to ensure the measurement, authentication and integrity of any privileged code which certainly include Cloudproxy components (like *linux_host*) as well as any dynamically linked libraries used by privileged applications. Cloudproxy applications should never trust “spinning” disks. All data written or read from these devices should be encrypted and integrity protected using keys protected by Cloudproxy. Finally, the disk to which images are swapped or paged should be encrypted (and either be physically secured, for example, by being in the same secure physical enclosure as the CPU and memory or cryptographically integrity protected). We achieve these goals using two Linux features: *initramfs* and *dmcrypt*. *Initramfs* is a ram based storage system which is loaded when the kernel is loaded and measured by tboot. We put all security critical code including *linux_host* and dynamic libraries used by Cloudproxy applications in *initramfs*; often we also include the actual Cloudproxy programs (although this is not strictly necessary). *Initramfs* is mounted as the root file system during boot and on most Linux systems, it is dismounted and replaced by the “system disk.” On Cloudproxy Linux, *initramfs* not dismounted and serves as the permanent system disk. All other disks are mounted under *initramfs* (and should not be trusted). This provides the complete control over system components and libraries we need. In addition, we encrypt the page file and swap devices using *dmcrypt*. *Dmcrypt* uses keys randomly generated at boot and we often require this “swap disk” be inside the secure physical cage protecting the computer (CPU and memory) running Cloudproxy.

Building InitRamfs

Everything upon which the security of your software relies must be measured, including basic system software and libraries you rely on. On Linux, we can protect and measure all that requires protection by incorporating it into a small, in memory, filesystem provided with the kernel boot image.

Initramfs is an in memory file system used by the Linux kernel. In most cases, initramfs is discarded late in the boot sequence of Linux in favor of the “real” mounted system disk but in our case, initramfs remains the root file system throughout the Base Linux Cloudproxy lifetime (other file systems are mounted under it). We modify “stock” initramfs in several ways. First, we get rid of any unnecessary functionality. Next we include the Cloudproxy components and, in some cases, even the Cloudproxy applications are included in initramfs. Lastly, we modify the stock scripts so that the “disk based” root file system is mounted under the initramfs rather than replacing it.

Detailed copy and paste instructions for initramfs are in the appendix.

Using Dmcrypt

Since most Linux systems employ virtual memory systems which temporarily store memory pages on disk, we must prevent secrets from being saved to a disk in a way that could be modified as programs are running or read after programs stop running. This would violate our security model. To do this, we employ dmcrypt which encrypts the disk used for paging. We also assume this disk is contained within the cage housing the Cloudproxy programs (since dmcrypt does not provide integrity protection). However, we *do not* require that the paging disk be protected before or after being used.

Detailed copy and paste instructions for configuring Linux to use dmcrypt are in the appendix.

Grub configuration

Grub is used to load the ACM, tboot, the OS (or hypervisor) and initramfs into main memory so that the measured boot can be carried out. Note that one can employ a grub configuration that can boot non Cloudproxy enabled images without harm.

This is done automatically by Tboot. Detailed copy and paste instructions for grub setup are in the appendix.

Installing and configuring a Cloudproxy supported KVM instance

We use KVM as the Cloudproxy hypervisor that can host isolated, measured Cloudproxy VMs.

Note that KVM Cloudproxy requires libvirt-dev and libtspi-dev be installed.

Detailed copy and paste instructions for KVM setup are in the appendix.

Installing and configuring a Cloudproxy supported TPM hosted Linux

In addition to a hypervisor rooted Cloudproxy stack, one could simply boot a Cloudproxy enabled Linux.

Detailed copy and paste instructions for Linux root setup are in the appendix.

Installing and configuring a Linux stacked on KVM

When using a hypervisor rooted Cloudproxy system, we need to provide Cloudproxy enabled VMs. Cloudproxy can support “stacked” systems that extend root protection to any layer of a software stack.

Detailed copy and paste instructions for configuring a stacked Linux Tao are in the appendix.

Installing and configuring Docker containers

Another stacked system protected by Cloudproxy is a Docker container within a Linux host (or stacked Linux host).

Detailed copy and paste instructions for configuring and running a stacked Docker Tao are in the appendix.

Host Certificates

Your TPM must have a signed certificate for its attest key. The underlying key is called an AIK in the case of TPM 1.2 and an Endorsement Certificate in the case of TPM 2.0. Consult the final appendix to provision this certificate for TPM 1.2 and TPM 2.0. In many cases, an Endorsement Certificate, signed by the manufacturer, is included in NVRam for TPM 2.0.

Domains

Domains and domain initialization was covered in [5] but when booting under TPM 1.2 and TPM 2.0, it is important to specify the proper PCR’s and, in the case of TPM 1.2, make sure the AIK was signed by the policy key using XXXX and, in the case of TPM 2.0 roots make sure the attest_service is running.

SimpleExample with TPM Tao’s

Complete instructions on running *simpleexample* under a TPM 1.2 Tao and a TPM 2.0 Tao appear in an appendix.

Considerations in deploying Cloudproxy

Key management

Key management supports the storage, distribution and critically, the rotation of symmetric content keys as well as establishing the “Program Key” for Cloudproxy programs (and hosts) and rotating the Program Key as keys expire or Cloudproxy programs are changed.

In most of our programs, keys and content types (like files) are objects and have domain-wide hierarchical universal names and epochs. Epochs are monotonically increased. For example, a file used by several Cloudproxy programs may contains customer information like customer names, addresses, etc. Since these are stored objects, they are encrypted and integrity protected and typically stored redundantly at several network accessible locations. The file /jlm/file/us-zone/customer-file-location1, epoch 2, type: file refers to the file with the indicated name; the epoch indicates the version of the file, different versions of the file are usually encrypted with different keys. These encryption keys also have domain-wide hierarchical universal names and epochs. Encryption keys protect other objects like keys or files. To decrypt a file, we need a chain of keys starting at the sealing key for the Cloudproxy program on a given node and terminating in the desired object (say a file). Each node of the chain consists of the name of the protector object, the protected objects and an encrypted blob consisting of the protected object encrypted (and integrity protected) by the protector key.

For example

/jlm/program-name/master-sealing-key, epoch 5, type: aes-128-gcm
/jlm/key/us-zone/customer-folder-keys, epoch 2, type: aes-128-gcm
/jlm/key/us-zone/customer-file-key, epoch 3, type: aes-128-gcm
/jlm/file/us-zone/customer-file-location1, epoch 2, type: file

Often domains will want to partition keys to provide resilience by *protection zones.* One common way to do this is to have different keys for different geographic locations. This is easily accomplished by reserving a level of the hierarchical object namespace for the zone name as is done above.

Although the foregoing name, epoch based mechanisms facilitate key rotation, it is not a complete solution. Let’s first consider, given this framework, how we might rotate keys.

There are two common circumstances where new keys are introduced into a Cloudproxy environment. First, keys should be rotated regularly as part of good cryptographic hygiene and second, when programs change, their measurements change which means they can no longer access sealed data for earlier versions of the program. In both cases, similar techniques can be used to carry out the key rotation.

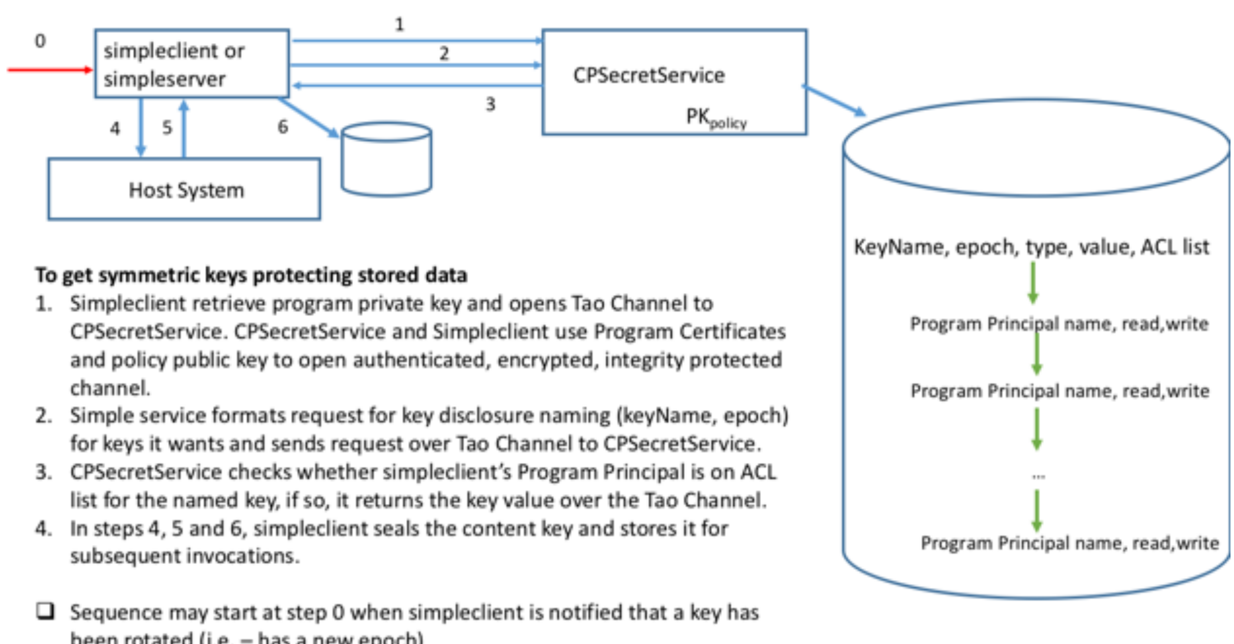
Key Rotation and software upgrades

See go/support/support_libraries /protected-objects and go/support/support_libraries/ rotation_support for support routines to maintain and encrypt content keys and build chains of keys like:

> Protector Object: /jlm/program-name/master-sealing-key, epoch 5, type: aes-128-gcm
> Protected Object: /jlm/key/us-zone-key, epoch 2, type: aes-128-gcm
> Encrypted Protected Object
>
> Protector Object: /jlm/key/us-zone-key, epoch 2
> Protected Object: /jlm/key/us-zone-key/customer-folder-key, epoch 3, type: aes-128-gcm
> Encrypted Protected Object
>
> Protector Object: /jlm/key/us-zone-key/ customer-folder-key, epoch 3
> Protected Object: /jlm/key/us-zone-key/customer-folder-key/customer-file-key, epoch 2, type: aes-128-gcm
> Encrypted Protected Object
>
> Protector Object: /jlm/key/us-zone/customer-file-key, epoch 3, type: aes-128-gcm
> Protected Object: /jlm/file/us-zone/customer-file-location1, epoch 2, type: file

Using a keystore

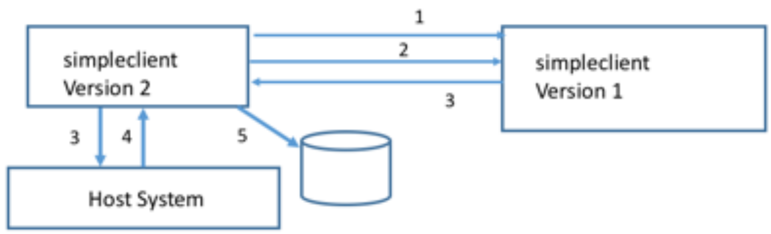
A sample keystore with the required functionality is implemented in `$CLOUDPROXY/go/support/infrastructure_support/CPSecretServer`. Needless to say, this program itself would likely be implemented as a cloudproxy program.



Getting content keys

Since keys can be fetched any time after startup and no pre-existing state is required (except for the Program key), key rotation is easy and is focused at the Secret Service. The secret service can also publish alerts when new keys are available. Upgrade when program versions change is also easy.   The new version of the program just gets keys from the server. The server is updated with new ACLs as new program versions become available.

Using policy key disclosure



To get symmetric keys protecting stored data from another program or version

1. Simpleclient, version 2 retrieves program private key and opens Tao Channel to Simpleclient, version 2. Simpleclient, version 2 and Simpleclient, version 1 use Program Certificates and policy public key to open authenticated, encrypted, integrity protected channel.
2. Simple service formats request for key disclosure naming (keyName, epoch) for keys it wants and sends request over Tao Channel to CPSecretService accompanied with policy-key signed certificate stating policykey says ProgramPrincipalName (simpleclient, version 2) can read /jlm/zone-us/customer-folder-key, epoch 2.
3. Simpleclient, version 1 transmits keys over tao channel.
4. 4, 5, 6, Simpleclient, version 2 seals retrieved key for subsequent use.

Getting content keys

See `$CLOUDPROXY/go/support/support_libraries/secret_disclosure_support` for sample code to construct and verify a policy-key signed secret-disclosure statement.

Policy-key says PrincipalName can-read /jlm/key/us-zone/customer-folder-key, epoch 2

A note on key rotation

Either the keystore mechanism or the secret disclosure mechanism can be used to protect other keys. For example, they can help disclose private signing keys to controlled groups. This allows a program to authenticate itself as a group or a standard Linux “service account,” making interoperability with “legacy” authorization systems easier.

Scale and deployment considerations

Redundancy, backup, audit, logging, dashboards, predicting problems, forensics

Just write your applications properly: simple, right?

While the Cloudproxy security model makes security reliance transparent and easily manageable, you still have to write your programs so that adversaries cannot exploit flaws in the programs you write (or the author of the VMM or BIOS wrote). This is not a trivial task but more programming tools, better programming languages and new techniques (such as proof-carrying-code) present plausible models for "much safer" software you can trust.

Oops: What to do if there's a gigantic breach

Although it is hopefully a rare event, a large scale failure (maybe caused by flaws in critical software) can be remedied simply by redistributing the application with a new policy key (after fixing the security flaws in this case).

Bugs and suggestions

Please post bugs and suggestions to the Github repository. Those suggestions will be treated as having been licensed under the Cloudproxy license.

Acknowledgements

Tom Roeder, Fred Schneider, Kevin Walsh and Sid Telang.

References

[1] Manferdelli, Roeder, Schneider, The CloudProxy Tao for Trusted Computing, http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf.
[2] CloudProxy Source code, http://github.com/jlmucb/cloudproxy. Kevin Walsh and Tom Roeder were principal authors of the Go version.
[3] TCG, TPM specs, http://www.trustedcomputinggroup.org/resources/tpm_library_specification
[4] Beekman, Manferdelli, Wagner, Attestation Transparency: Building secure Internet services for legacy clients**.** AsiaCCS, 2016.
[5] Manferdelli, CloudProxy Nuts and Bolts.
[6] Intel, The MLE Development Guide

Appendix 1 - Supported Hardware

Intel or AMD CPU’s and chipsets with secure extensions that support SKINIT. In addition, there must be a physical TPM on the motherboard or in firmware. Most new Intel CPU’s have firmware TPM’s so most new hardware is “automatically” Cloudproxy enabled.

Dell Omniplex
HP xxx laptops
Tom’s laptop
NUC (be sure it supports SMX)

Appendix 2 - TPM 1.2 Initialization and Operation (Optional)

Trousers supports some TPM initialization operations but we have tried to structure installation so that you do not need to use Trousers in simple Cloudproxy deployments. As a result, this section is informational.

Clearing and taking ownership of TPM 1.2 with Trousers

tpm_clear --force
Tspi_TPM_ClearOwner failed: 0x00000007 - layer=tpm, code=0007 (7), TPM is disabled
We can see that the TPM is disabled, which is why we can't clear it. This can happen if we forget to actually enable the TPM in BIOS. The first thing to do would be to actually enable the TPM in BIOS. But if the TPM has been initialized before, we would receive the output that can be seen below:

tpm_clear --force
TPM Successfully Cleared. You need to reboot to complete this operation. After reboot the TPM will be in the default state: unowned, disabled and inactive.
This would require us to reboot the computer for changes to take effect. When clearing the TPM we'll return it to the default state, which is unowned, disabled and inactive, as already mentioned. To enable the TPM afterwards, we need the owner password. But since the TPM owner has been cleared, there is no owner password and we can set a new one without entering the old one. We can also receive an error like the following:

tpm_clear --force
Tspi_TPM_ClearOwner failed: 0x0000002d - layer=tpm, code=002d (45), Bad physical presence value

DO NOT SET ANY PASSWORD for the TPM.
tpm_takeownership -z -y
If we later want to change either of the commands, we can do it with the tpm_changeownerauth command. If we pass the –owner argument to the tpm_changeownerauth command we'll be changing the administration password and if we pass the –srk into the tpm_changeownerauth command we'll be changing the SRK password. We can see the example of both commands in the output below:

tpm_changeownerauth --owner
Enter owner password:
Enter new owner password:
Confirm password:

tpm_changeownerauth --srk
Enter owner password:

Enter new SRK password:
Confirm password:

There are 5 keys in TPM:

TPM Endorsement Key (EK): This key is created by the manufacturer and cannot be removed. Sometimes it can be changed by the owner of the computer.
TPM Storage Key (SRK): Is the 2048 bit RSA key created when configuring the ownership. This key is stored inside the chip and can be removed. The key is used to encrypt the Storage Key (SK) and Attestation Identity Key (AIK).# tpm_setenable --enable
Enter owner password:
Disabled status: false

tpm_setactive
Enter owner password:
Persistent Deactivated Status: false
Volatile Deactivated Status: false
There are usually two Endorsement Keys (EK): the public and private one. The private key is always stored at the TPM and cannot even be seen by anyone, while the public key can be displayed with the tpm_getpubek command.

tpm_getpubek
Tspi_TPM_GetPubEndorsementKey failed: 0x00000008 - layer=tpm, code=0008 (8), The TPM target command has been disabled
Enter owner password:

```
Public Endorsement Key:
  Version:   01010000
  Usage:     0x0002 (Unknown)
  Flags:     0x00000000 (!VOLATILE, !MIGRATABLE, !REDIRECTION)
  AuthUsage: 0x00 (Never)
  Algorithm:         0x00000020 (Unknown)
  Encryption Scheme: 0x00000012 (Unknown)
  Signature Scheme:  0x00000010 (Unknown)
  Public Key:
        a350b3a3 3edddc30 06248f4f 5d3eb80a 34fcbea0 83dde002 8dffa703 e116f8b0
        eb1962ee a65998b3 384aeb6e 85486be9 0316a6ca a189a5ba 2217b2a2 9da014db
        dfbe7731 fb675e7a 438c4775 deea54fb 0c75de5d ba961950 3eda4555 d27a9a30
        e94d39d0 a4ea314d a70eaf08 e49dd354 d57ed34d 234220d9 604471a9 86173050
        9ff9b0e5 b65cb4b5 5f46a7f9 4378bd7e 8c61b91b ad312974 fef5d70f 84f4484f
        e5c95300 0eef76f2 1667443f dc2fa82e 351d945e 6b5f75e8 828d010f 61541552
    [...]
```

Troubleshooting TPM 1.2

tpm-takeownership fails: tpm-takeownership assumes the owner of the device is not set. Reset the TPM through BIOS as explained in the main TPM 1.2 section to reset ownership.

tpm-takeownership not found: Make sure $GOPATH/bin is in your PATH environment variable.

Using tpm-clear: is NOT recommended for resetting the device or clearing ownership, and it is known to leave the TPM in a mucked state from where tpm-takeownership does not work. Reset the TPM through BIOS as explained in the main TPM 1.2 section to reset the device/ownership.

Appendix 3 - Tboot Launch Policy for TPM 1.2 (Optional)

You may build a Launch Control Policy (LCP) which controls what can be booted. If you have no LCP, the default policy will be applied and this should work, so this appendix, like the last is informational only.

Trousers and trousers-devel packages must already be installed in order to build lcptools.

Next you must modify grub so that you can choose the tbooted linux from the boot window.

If the verified launch policy is non-empty, it is extended into PCR 17 by default. Subsequent loaded modules are extended into PCR 18 by default (Check this!). This behavior can be changed with the VLP.

The Grub(2) configuration file is usually in /boot/grub and is called grub.cfg. It is updated automatically when a kernel is updated or when you run update-grub.

Tboot module must be added as the 'kernel' in the grub.conf file.

Note that the Lenovo T410 is known not to work with the Intel IOMMU; it will not successfully boot with TBOOT.

The final grub configuration file will look something like:

```
menuentry 'Ubuntu Linux 3.0.0-16-generic with TXT' --class ubuntu --class gnu-linux --
class gnu --class os {
        recordfail
        set gfxpayload=text
        insmod gzio
        insmod part_msdos
        insmod ext2
        set root='(hd0,msdos5)'
        search --no-floppy --fs-uuid --set=root 8ab78657-8561-4fa8-af57-bff736275cc6
        echo 'Multiboot'
        multiboot /boot/tboot.gz /boot/tboot.gz logging=vga,memory,serial
        echo 'Linux'
        module   /boot/vmlinuz-3.0.0-16-generic /boot/vmlinuz-3.0.0-16-generic
root=UUID=8ab78657-8561-4fa8-af57-bff736275cc6 ro   splash vt.handoff=7 intel_iommu=on
        echo 'initrd'
        module /boot/initrd.img-3.0.0-16-generic /boot/initrd.img-3.0.0-16-generic
        echo 'sinit'
        module /boot/sinit51.bin /boot/sinit51.bin
```

Modify grub.conf to load the policy data file:

Edit grub.conf and add the following:
    module /list.data

where you should use the path to this file.
Copy sinit into /boot and change run grub.conf then run update-grub.
Check the /boot directory to make sure tboot.gz is there.

The utility txt-stat in the utils subdirectory of tboot, can be used to view the DRTM boot log.

Reported problems with TPM 1.2

The TPM, driver/char/tpm/tpm.c, depends on TPM chip to report timeout values for timeout_a, b, and d. The Atmel TPM in Dell latitude 6430u, reports wrong timeout values (10 ms each), instead of TCG specified (750ms, 2000ms, 750ms, 750ms for timeout_a/b/c/d respectively). You can fix the driver code and it will work.

A permissive Tboot policy is required for the non-hypervisor solutions. In this case, the system will boot whatever it finds, but it will still perform measured launch, and clients interacting with the system will still be able to check that the right software was booted by checking the PCRs. Once the hypervisor is written, a more restrictive launch control policy is possible and maybe desirable.

Put 11_tboot in /etc/grub.d

Information on PCR Usage on TPM 1.2

PCR 17 will be extended with the following values (in this order):

- The values as documented in the MLE Developers Manual
- SHA-1 hash of: tboot policy control value (4 bytes) SHA-1 hash of tboot policy (20 bytes) : where the hash of the tboot policy will be 0s if TB_POLCTL_EXTEND_PCR17 is clear

PCR 18 It will be extended with the following values (in this order):

- SHA-1 hash of tboot (as calculated by lcp_mlehash)
- SHA-1 hash of first module in grub.conf (e.g. Xen or Linux kernel)

PCR * : tboot policy may specify modules' measurements to be extended into PCRs specified in the policy
The default tboot policy will extend, in order, the SHA-1 hashes of all modules (other than 0) into PCR 19.

The LCP consists of policy elements.

To specify launch policy via a list of hashes:

```
0.  sudo bash
1.  lcp_mlehash -c "command line for tboot from grub.cfg" /boot/tboot.gz > mle_hash
  - the command line in this case is the string from /boot/grub/grub.cfg
  - after multiboot tboot.gz, e.g., "logging=vga,memory,serial"
  - copy and paste:
./lcp_mlehash -c "logging=vga" /boot/tboot.gz>mle_hash
2.  lcp_crtpolelt --create --type mle --ctrl 0x00 --minver 17 --out mle.elt mle_hash
  - copy and paste
./lcp_crtpolelt --create --type mle --ctrl 0x00 --minver 17 --out mle.elt mle_hash
```

```
3.  lcp_crtpollist --create --out list_unsig.lst mle.elt pconf.elt
  - copy and paste
./lcp_crtpollist --create --out list_unsig.lst mle.elt pconf.elt
4.  lcp_crtpol2 --create --type list --pol list.pol --data list.data list_unsig.lst
  - copy and paste
./lcp_crtpol2 --create --type list --pol list.pol --data list.data list_unsig.lst
5.  cp list.pol /boot
  - copy and paste
cp owner_list.data /boot
```

Next create a verified Launch policy:

```
1.  tb_polgen/tb_polgen --create --type nonfatal vl.pol
  - copy and paste
../tb_polgen/tb_polgen --create --type nonfatal vl.pol
2.  tb_polgen/tb_polgen --add --num 0 --pcr 18 --hash image
    --cmdline "the command line from linux in grub.conf"
    --image /boot/vmlinuz-2.6.18.8
    vl.pol
  - copy and paste
../tb_polgen/tb_polgen --add --num 0 --pcr 18 --hash image --cmdline "root=UUID=cf6ae6b5-
abb5-4d5d-b823-bd798a0621de ro quiet splash $vt_handoff" --image /boot/vmlinuz-3.5.0-23-
generic vl.pol
3.  tb_polgen/tb_polgen --add --num 1 --pcr 19 --hash image
    --cmdline ""
    --image /boot/initrd-2.6.18.8
    vl.pol
  - copy and paste
../tb_polgen/tb_polgen --add --num 1 --pcr 19 --hash image --image /boot/initrd-3.5.0-23-
generic vl.pol

If any mle can be launched:
./lcp_crtpol2 --create --type any --pol any.pol

Define tboot error TPM NV index:
1.  lcptools/tpmnv_defindex -i 0x20000002 -s 8 -pv 0 -rl 0x07 -wl 0x07
    -p TPM-password
  - copy and paste
../lcptools/tpmnv_defindex -i 0x20000002 -s 8 -pv 0 -rl 0x07 -wl 0x07

Define LCP and Verified Launch policy indices:
1.  lcptools/tpmnv_defindex -i owner -s 0x36 -p TPM-owner-password
  - copy and paste
../lcptools/tpmnv_defindex -i owner -s 0x36 -p TPM-owner-password
2.  lcptools/tpmnv_defindex -i 0x20000001 -s 256 -pv 0x02 -p TPM-owner-password
  - copy and paste
../lcptools/tpmnv_defindex -i 0x20000001 -s 256 -pv 0x02 -p TPM-owner-password

Write LCP and Verified Launch policies to TPM:
(modprobe tpm_tis; tcsd;)
1.  lcptools/lcp_writepol -i owner -f [any.pol|list.pol] -p TPM-password
  - copy and paste
../lcp_writepol -i owner -f list.pol -p <ownerauth password>
2.  If there is a verified launch policy:
    lcptools/lcp_writepol -i 0x20000001 -f vl.pol -p TPM-password
  - copy and paste
../lcptools/lcp_writepol -i 0x20000001 -f vl.pol -p TPM-password
```

Use lcp_crtpollist to sign the list:

1. openssl genrsa -out privkey.pem 2048

2. openssl rsa -pubout -in privkey.pem -out pubkey.pem
3. cp list_unsig.lst list_sig.lst
4. lcp_crtpollist --sign --pub pubkey.pem --priv privkey.pem --out list_sig.lst

Use openssl to sign the list:

1. openssl rsa -pubout -in privkey.pem -out pubkey.pem
2. cp list_unsig.lst list_sig.lst
3. lcp_crtpollist --sign --pub pubkey.pem --nosig --out list_sig.lst
4. openssl genrsa -out privkey.pem 2048
5. openssl dgst -sha1 -sign privkey.pem -out list.sig list_sig.lst
6. lcp_crtpollist --addsig --sig list.sig --out list_sig.lst

Appendix 4 - TPM 2.0 Installation and Operation

The interface to TPM 2.) is via BIOS, unlike TPM 1.2. TPM 2's have pretty severe limitations on the number of open handles.

Troubleshooting TPM 2.0

Too many open handles: Run

    tpm2_util.exe –command=Flushall

TPM2 Error code 0x9a2:
This error code corresponds to 'bad authorization' and is returned when a command has incorrect authorization values. If you encounter this when using CloudProxy, it means your TPM is not provisioned properly, and you would need to clear the TPM.

TPM2 Error code 0x902:
This error code is returned when the TPM is out of memory for creating new objects. Run the TPM2_FlushContext command for all open handles (which can be obtained by running the TPM2_GetCapability command).

TPM2_Clear on the Intel NUC 5i5MYHE:
Clearing the TPM has the effect of setting all authorization values to *emptyAuth* (i.e. empty string as password). Clearing a TPM can be done through the maintenance screen of the BIOS, which is accessed by booting up the machine with the yellow BIOS jumper removed, and pressing the Esc key when the startup BIOS screen is shown.

Appendix 5 - Tboot

To build tboot you first need to install trousers and libtspi-dev.

Installing TBOOT

1. Download the source tarball from https://sourceforge.net/projects/tboot/
2. In the untarred source directory run:
   `make`
3. In the same directory as above, run
   `sudo make install`
4. Check the `/boot` directory contains the `tboot.gz` file
5. Download the SINIT AC module corresponding to your processor from https://software.intel.com/en-us/articles/intel-trusted-execution-technology
6. Copy the SINIT `*.BIN` file to your `/boot` directory
7. To start using tboot, you need to update your bootloader configuration. If using grub2, run
   `sudo update-grub`
   And check that /boot/grub/grub.cfg now contains an entry for tboot which looks like:

```
menuentry 'Ubuntu GNU/Linux, with tboot 1.9.4 and Linux 4.4.0-24-generic' --class ubuntu --class gnu-linux --class gnu --class os --class tboot {
        insmod multiboot2
        insmod part_gpt
        insmod ext2
        set root='hd0,gpt2'
        if [ x$feature_platform_search_hint = xy ]; then
          search --no-floppy --fs-uuid --set=root --hint-bios=hd0,gpt2 --hint-efi=hd0,gpt2 --hint-baremetal=ahci0,gpt2  1df958d9-c01d-43d8-a1b2-18d022843d3f
        else
          search --no-floppy --fs-uuid --set=root 1df958d9-c01d-43d8-a1b2-18d022843d3f
        fi
        echo    'Loading tboot 1.9.4 ...'
        multiboot2      /boot/tboot.gz logging=serial,memory
        echo    'Loading Linux 4.4.0-24-generic ...'
        module2 /boot/vmlinuz-4.4.0-24-generic root=UUID=1df958d9-c01d-43d8-a1b2-18d022843d3f ro quiet splash intel_iommu=on noefi
        echo    'Loading initial ramdisk ...'
        module2 /boot/initrd.img-4.4.0-24-generic
        echo    'Loading sinit 5th_gen_i5_i7_SINIT_79.BIN ...'
        module2 /boot/5th_gen_i5_i7_SINIT_79.BIN
}
```

8. If using tboot with cloudproxy, you need to adjust your bootloader configuration differently (to add the initramfs). Instructions to do so can be found in a subsequent appendix.

More information about Tboot can be found in the README in the Tboot source directory.

Tboot will copy and alter the e820 table provided by GRUB to "reserve" its own memory plus the TXT memory regions. These are marked as E820_UNUSABLE or E820_RESERVED so that the patched Xen code can prevent them from being assigned to dom0. The e820 table is not altered if the measured launch fails for any reason.

Appendi 6 - Initramfs

1. **Download and unpack Linux kernel and busybox source**
   We use Busybox to provide a basic set of Unix utilities to use on the VM instance.

```
mkdir /tmp/tiny_linux
   cd /tmp/tiny_linux
   curl https://www.kernel.org/pub/linux/kernel/\
   v4.x/linux-4.0.3.tar.xz | tar xJf -
   curl https://busybox.net/downloads/busybox-1.23.2.tar.bz2 \
| tar xjf -
```

2. **Build busybox**

```
mkdir -pv ../obj/busybox-x86
make O=../obj/busybox-x86 defconfig
   make O=../obj/busybox-x86 menuconfig
```

   The last command opens an interactive menu. We will select to build BusyBox statically to ensure it can run on the VM instance.

```
-> Busybox Settings
  -> Build Options
[ ] Build BusyBox as a static binary (no shared libs)
```

   Select the above option, save your selection and quit the interactive menu.
   Next we build Busybox as shown below.

```
   cd ../obj/busybox-x86
   make -j2
   make install
```

3. **Build initrd image**
   Next we build the init RAM disk image.

```
mkdir -p /tmp/tiny_linux/initramfs/x86-busybox
cd /tmp/tiny_linux/initramfs/x86-busybox
mkdir -pv {bin,sbin,etc,proc,sys,usr/{bin,sbin}}
cp -av $TOP/obj/busybox-x86/_install/*
```

   We will need an init script. The following minimal script works.

```
#!/bin/sh
mount -t proc none /proc
```

```
mount -t sysfs none /sys
mount -t devtmpfs -o mode=0755 udev /dev

exec /bin/sh
```

Save the above script in a file called init and make it executable.

```
chmod +x init
```

Finally we archive and compress this directory to create our initrd image.

```
find . -print0 \
    | cpio --null -ov --format=newc \
    | gzip -9 > /tmp/tiny_linux/obj/initramfs-busybox-x86.cpio.gz
```

4. **Build linux kernel.**

```
cd /tmp/tiny_linux/linux-4.0.3
make O=../obj/linux-x86-basic x86_64_defconfig
make O=../obj/linux-x86-basic kvmconfig
make O=../obj/linux-x86-basic -j2
```

The kernel image can be found at `/tmp/tiny_linux/obj/linux-x86-basic/arch/x86_64/boot/bzImage`

Appendix 7 - Dmcrypt

Dmcrypt is a standard Linux component, we use it to encrypt page tables so that there is no sensitive information on disks. As a result, while the system paging disk should be in the “cage” during operation, these disks do not contain sensitive information and do not have to be specially handled after operation. Disks do not have to be wiped.

fdisk to make partitions (/dev/sda)
mkfs.ext2 to make file system
change to single user mode
grub-install root-directory=/dev/sda
CRYPTDISKS_ENABLE=Yes in /etc/defaults/cryptdisks

```
/etc/crypttab
swap /dev/hda1 /dev/urandom swap
/etc/fstab
/dev/swapper/swap none
swap sw,pri=1 0 0
```

For swap:

```
swapoff -a
cryptsetup [-c aes -h sha256] -s 128 -d /dev/urandom create swap
/dev/sda1
mkswap /dev/mapper/swap
swapon /dev/mapper/swap
```

Shell script prototype

```
#
#   For debian, first do the following:
#
#   Change to single user mode
#       cat /proc/swaps   (to find out swap device)
#       /sbin/runlevel    (to find runlevel)
#       /sbin/telinit 1   (to go single user)
#       ctrl-alt-f7 puts Linux into console mode.
#
#   /etc/defaults/cryptdisks
#       CRYPTDISKS_ENABLE=Yes
#   /etc/crypttab
#       swap /dev/sda5 /dev/urandom swap
#   /etc/fstab
#       /dev/mapper/swap none
#       swap sw,pri=1 0 0
#   /etc/crypttab
#       swap /dev/sda5 /dev/urandom cipher=aes-cbc-essiv:sha256,size=256,hash=sha256,swap
#   /etc/fstab
#       /dev/mapper/cryptoswap none swap sw 0 0
#
#   /dev/sda5 is swap device
#
# Ref: The whole disk Nov, 2006, Linux-magazine.com, Michael Nerb
#
```

```
swapoff -a
cryptsetup -c aes -h sha256 -s 128 -d /dev/urandom create swap /dev/sda5
mkswap /dev/mapper/swap
swapon /dev/mapper/swap
#
# To remove:
#   swapoff /dev/mapper/swap
#   cryptsetup [luks] remove swap
```

Appendix 8 - Grubby, Grub, Grub (Optional)

This appendix is informational, tboot does all of this automatically in the normal course of events.

The new tboot module must be added as the 'kernel' in the grub.conf file. The existing 'kernel' entry should follow as a 'module'. The SINIT ACM module must be added to the grub.conf boot config as the last module, for example:

```
root (hd0,1)
     kernel /tboot.gz logging=serial,vga,memory
     module /vmlinuz-2.6.18 root=/dev/VolGroup00/LogVol00 ro
     module /initrd-2.6.18.img
     module /Q35_SINIT_17.BIN
```

The kernel module is tboot. The next is the Linux (possibly KVM enabled) kernel. Next is the initramfs and finally the ACM module.

GRUB2 does not pass the file name in the command line field of the multiboot entry (module_t::string). Since the tboot code is expecting the file name as the first part of the string, it tries to remove it to determine the command line arguments, which will cause a verification error. The "official" workaround for kernels/etc. that depend on the file name is to duplicate the file name in the grub.config file like below:

```
menuentry 'Xen w/ Intel(R) Trusted Execution Technology' {
        recordfail
        insmod part_msdos
        insmod ext2
        set root='(/dev/sda,msdos5)'
        search --no-floppy --fs-uuid --set=root 4efb64c6-7e11-482e-8bab-
07034a52de39
        multiboot /tboot.gz /tboot.gz logging=vga,memory,serial
        module /xen.gz /xen.gz iommu=required dom0_mem=524288
com1=115200,8n1
        module /vmlinuz-2.6.18-xen /vmlinuz-2.6.18-xen
root=/dev/VolGroup...
        module /initrd-2.6.18-xen.img /initrd-2.6.18-xen.img
        module /Q35_SINIT_17.BIN
    }
```

Appendix 9 - Setting up a Cloudproxy enabled KVM host

In this section we will see how to set up a Cloudproxy enabled KVM host, which can launch independent Cloudproxy VMs (i.e. VMs with a stacked Cloudproxy host running)

The root Tao of our CloudProxy KVM host will be a “soft” Tao: i.e. a dummy Tao based on a keyset that is stored on disk, encrypted and integrity protected by a password.

CoreOS Instances

1. **Generate soft Tao.**
   We set up the Tao domain directory and the host directory, where the domain and host configuration information will be stored. We copy in a domain template which we will use in this example and setup the soft Tao which generates a password protected keyset in the linux_tao_host directory.

   ```
   mkdir /tmp/domain
   cd /tmp/domain
   mkdir linux_tao_host
   cp $GOPATH/src/github.com/jlmucb/\
      cloudproxy/go/apps/simpleexample/SimpleDomain/\
      domain_template.simpleexample .
   tao domain newsoft \
      -config_template domain_template.simpleexample \
      -soft_pass xxx ./linux_tao_host
   ```

2. **Initialize domain**
   We initialize the domain which will create policy keys, saved in the domain directory and protected by the password. It will also create a tao.config file which stores all the domain configuration information and is used by the host to load the domain.

   ```
   tao domain init \
         -tao_domain . \
         -config_template domain_template.simpleexample \
         -pass xxx
   ```

3. **Create linux_host image**
   We statically build a linux host image (so that it is executable in the VM) and create a tarball of the linux_host binary and the host/domain configuration information it needs to run. When launching a VM, the KVM Cloudproxy unpacks this, mounts it on the VM and runs linux_host as a stacked Cloudproxy host.

```
CGO_ENABLED=0 go install -a \
   -installsuffix nocgo \
    github.com/jlmucb/cloudproxy/go/apps/linux_host
cp $GOPATH/bin/linux_host .
tar -C . -czf linux_host.img.tgz $(ls .)
```

4. **Get CoreOS image**
   Get the latest stable image of CoreOS.

```
curl -G http://stable.release.core-os.net/amd64-usr/\
   current/coreos_production_qemu_image.img.bz2 > \
   coreos_production_qemu_image.img.bz2
bunzip2 coreos_production_qemu_image.img.bz2
```

5. **Setup ssh-agent**
   We generate a key pair and setup ssh-agent for SSHing into the VM instance.

```
ssh-keygen -t rsa -f ./id_rsa
ssh-add id_rsa
```

6. **Initialize and start host**
   We initialize and start the KVM Cloudproxy host

```
tao host init \
   -root \
   -tao_domain . \
   -pass xxx \
   -hosting kvm_coreos \
   -kvm_coreos_img coreos_production_qemu_image.img \
   -kvm_coreos_ssh_auth_keys id_rsa.pub

tao host start \
   - tao_domain . \
   - host ./linux_tao_host
```

7. **Launch hosted program (i.e. VM)**
   We launch a Cloudproxy VM through the Cloudproxy KVM host. We create another temporary directory where the linux_host tarball is unpacked and mounted on the VM and run from.

```
mkdir another_tmp
tao_launch run \
   -tao_domain . \
   -host ./linux_tao_host \
```

```
-verbose kvm_coreos:linux_host.img.tgz \
-- linux_host.img.tgz another_tmp 2222
```

Launching a VM with custom kernel through KVM Cloudproxy

1. **Generate soft Tao.**
   We set up the Tao domain directory and the host directory, where the domain and host configuration information will be stored. We copy in a domain template which we will use in this example and setup the soft Tao which generates a password protected keyset in the linux_tao_host directory.

```
mkdir /tmp/domain
cd /tmp/domain
mkdir linux_tao_host
cp $GOPATH/src/github.com/jlmucb/\
   cloudproxy/go/apps/simpleexample/SimpleDomain/\
   domain_template.simpleexample .
tao domain newsoft \
   -config_template domain_template.simpleexample \
   -soft_pass xxx ./linux_tao_host
```

2. **Initialize domain**
   We initialize the domain which will create policy keys, saved in the domain directory and protected by the password. It will also create a tao.config file which stores all the domain configuration information and is used by the host to load the domain.

```
tao domain init \
      -tao_domain . \
      -config_template domain_template.simpleexample \
      -pass xxx
```

3. **Initialize and start host**
   We initialize and start the KVM Cloudproxy host

```
tao host init \
   -root \
   -tao_domain . \
   -pass xxx \
   -hosting kvm_custom \

tao host start     -tao_domain .
```

4. **Copy Kernel image and initrd image to domain directory**
   Replace the paths in the following lines with the path tyour kernel and initrd image.

```
cp /tmp/tiny_linux/obj/linux-x86-basic/arch/x86_64/boot/bzImage .
cp /tmp/tiny_linux/obj/initramfs-busybox-x86.cpio.gz .
```

5. **Launch hosted program (i.e. VM)**
   We launch a Cloudproxy VM through the Cloudproxy KVM host.

```
tao_launch run \
   -tao_domain . \
   -verbose kvm_custom: \
   -- bzImage initramfs-busybox-x86.cpio.gz 2222
```

Appendix 10 - SimpleExample with a TPM 1.2 Tao

Here we go through the procedure to run *simpleexample* (as described in Cloudproxy Nuts and Bolts) under a Linux whose root host is a TPM 1.2 rather than a "Soft Tao."   In addition, while the *simpleexample* employed a *simpledomainservice*, here we use a domain service which fully verifies the attestation before issuing Program Certificates.

We assume you've carried out the *simpleexample* tutorial including the following steps:

As root:

```
mkdir  /Domains
chown yourusername /Domains
```

As yourusername

```
cd $CLOUDPROXY/go
      go install …
cd $CLOUDPROXY/go/apps/simpleexample/SimpleDomain
cp ~/bin/copybins
```

To run simpleexample under a TPM 1.2 Tao, first (as whatever your user account is),

```
./initsimpleexampletpm1
```

This will

As root

```
./inittpm1domain
./runalltpm1simple
```

Appendix 11 - SimpleExample with a TPM 2.0 Tao

\

Here we go through the procedure to run *simpleexample* (as described in Cloudproxy Nuts and Bolts) under a Linux whose root host is a TPM 2.0 rather than a "Soft Tao."   The example in Cloudproxy Nuts and Bolts*,* employed *simpledomainservice*, which did not compare the program requesting the Program Certificate against a list of authorized programs as would be required to meet Cloudproxy's security goals. Here we show how to run *simpleexample* either with the *simpledomainserver* or *domain_server*. *domain_server* verifies the attestation before issuing Program Certificates so it a much more realistic example.

In our example, the endorsement certificate is signed by the policy key but in real deployment one might rely of endorsement certificates provided by the TPM vendor. These certificates can often be found in NvRam in compliant TPMs.

We assume you've carried out the *simpleexample* tutorial including the following steps:

As root:

```
mkdir  /Domains
chown yourusername /Domains
```

As yourusername

```
cd $CLOUDPROXY/go
      go install …
cd $CLOUDPROXY/go/apps/simpleexample/SimpleDomain
cp ~/bin/copybins
```

We also assume you have removed unneeded files and killed previous instances of Cloudprox programs as described in the *simpleexample* instructions in "Nuts and Bolts."

Having done the foregoing, to run simpleexample under a TPM 2.0 Tao, as whatever your user account is, run

```
./initsimpleexampletpm2
```

This will construct the endorsement cert, create the storage hierarchy for simpleexample under TPM2 and … Then, as root

```
./inittpm2domain
./runalltpm2simple
```

This will run the entire example with the *SimpleDomainServer*, which doesn't check the Tao Principal Name to ensure the program requesting the Program is one of the programs "trusted" in the domain. Another program, *domain_server* does these checks. To run this full example, first, you must create a file in */Domains/domain.simpleexampletpm2* called *TrustedEntities* naming the authorized programs. Create the file `/Domains/domain.simpleexampletpm2/TrustedEntities` and add two lines:

```
TrustedEntity: “Tao-name-of-simpleserver”
TrustedEntity: “Tao-name-of-simpleclient”
```

You can copy the Tao Principal Names of these programs from the output of *runalltpm2simple*, you will need to escape and quotes in the names by preceding the quote with a \.

This will look something like:

```
TrustedEntity:
“tpm2([6b6b6304dd984fc9af91569e5a25366eacf66fb8821830665f93e94aaf5943a2
]).PCRs(\"17,18\",\"ffffffffffffffffffffffff\"...).TrivialGuard(\"Liber
al\").Program([a04ae93feaca725df5c48939a70812df4f065a14d4b27e1618976810
aac2e7a7]).PolicyKey([1bbf737f810e672aee79b82d946f945fe7472dddc982bb17d
f21f44da1fa50d3])”
TrustedEntity:
“tpm2([6b6b6304dd984fc9af91569e5a25366eacf66fb8821830665f93e94aaf5943a2
]).PCRs(\"17,18\",\"ffffffffffffffffffffffffffffffffffffffff,ffffffffff
ffffffffffffffffffffffffffffff\").TrivialGuard(\"Liberal\").Program([f5
e895bf4f667077cc778002ba4d9aa64c960115732d065d009b58194392e34b]).Policy
Key([1bbf737f810e672aee79b82d946f945fe7472dddc982bb17df21f44da1fa50d3])
”
```

Next, as root, run:

```
./inittpm2domain
./runalltpm2full
```

Appendix 12 - SimpleExample in VM using stacked host, KVM host under TPM 2.0 Tao

Here we go through the procedure to run *simpleexample* (as described in Cloudproxy Nuts and Bolts) under a Linux in a VM hosted by KVM. The KVM host employs a root host that is a TPM 2.0 host rather than a "Soft Tao." As with the prior two examples, while the *simpleexample* employed a *simpledomainservice*, here we use a domain service which fully verifies the attestation before issuing Program Certificates. This is our first "non-trivial" example of a stacked Tao.

Build the KVM instance

Provision the TPM endorsement certificate

Build the Linux VM

Run two Linux VM's, one with with the simpleclient and one with simpleserver.